ANALYZING ALGEBRAIC CURVES BY CLUSTER COMPUTING

Wolfgang Schreiner, Christian Mittermaier, Franz Winkler
Research Institute for Symbolic Computation (RISC-Linz)
Johannes Kepler University, Linz, Austria
http://www.risc.uni-linz.ac.at
FirstName.LastName@risc.uni-linz.ac.at*

Abstract We describe a parallel solution to the problem of resolving non-ordinary singularities of a plane algebraic curve. The original sequential program is implemented in the software library CASA on top of the computer algebra system Maple. The new parallel version is based on Distributed Maple, a distributed programming extension written in Java. We evaluate the performance of the program in a cluster environment and compare it to that on a massively parallel multiprocessor.

Keywords: Computer algebra, Maple, parallel algorithm, cluster, Java.

1. INTRODUCTION

We describe a parallel solution to the problem of resolving non-ordinary singularities of a plane algebraic curve. The starting point of our work is the library CASA (computer algebra software for constructive algebraic geometry) which has been developed since 1990 by various researchers under the direction of the third author [5]. CASA is based on the computer algebra system Maple.

In order to parallelize a number of CASA functions, the first author has developed "Distributed Maple", a system that allows to implement parallel algorithms in Maple and to execute them in any networked environment [6, 7]. Its core is a scheduler program written in Java which is in charge of coordinating the activities of the Maple kernels running on various machines. The system

*Supported by the FWF grants P11160-TEC (HySaX) und SFB F013/F1304.

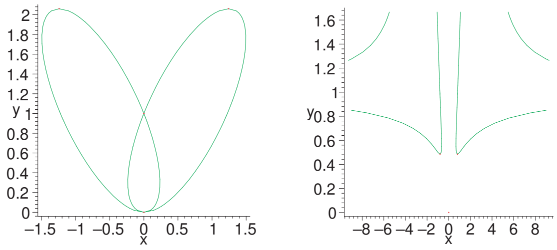


Figure 1.1 The Tacnode Curve and its Transformed Version

has evolved from our own experience in the development of parallel computer algebra environments [3] and from related work of other researchers [9, 2, 1].

On the basis of Distributed Maple, we have parallelized several algorithms from algebraic geometry, mostly in the frame of the diploma thesis of the second author [4]. In this paper, we present one of these parallel solutions.

2. RESOLUTION OF SINGULARITIES

The basic objects of the software library CASA are algebraic sets represented e.g. as systems of polynomial equations [5]. Algebraic sets represented by bivariate polynomials model plane curves; an important problem is to determine whether such a curve has a *rational parameterization* [8], i.e., whether there exist rational functions $X(t), Y(t)$ such that, for almost all parameters $t \in \mathbb{R}$, the point $(X(t), Y(t))$ is on the curve and, for almost every point (x, y) on the curve, there exists some $t \in \mathbb{R}$ such that $(x, y) = (X(t), Y(t))$.

A curve has such a parameterization only if $(d-1)(d-2) = \sum_{i=1}^{n} m_i(m_i - 1)$ where d is degree of the polynomial p defining the curve and n is the number of curve singularities with multiplicities $m_1, \ldots, m_n$, respectively. A singularity is a curve point P such that both partial derivatives $\frac{\partial p}{\partial x}$ and $\frac{\partial p}{\partial y}$ vanish at P; the multiplicity of P is the largest m such that all derivatives $\frac{\partial^{u+v} p}{\partial x^u \partial y^v}$ with $u + v < m$ vanish at P; the curve then has at most m tangents at P.

Above decision criterion only holds, if every singularity is ordinary, i.e., if its multiplicity m denotes m tangents. If a curve C has a non-ordinary singularity, we may however transform it into some C_1 where the non-ordinary singularity is resolved into ordinary ones (see the right curve in Figure 1.1 that resolves the non-ordinary singularity $(0, 0)$). By repeating this transformation, we thus compute a sequence $C_1, C_2, \ldots, C_n$ such that C_n has only ordinary singularities and we may decide from C_n whether C has a rational parameterization [8].

Each C_i is given in projective space, i.e., it is defined by a homogeneous polynomial $p(x, y, z)$. A part of the transformation is the computation of all singularities of p of the form $(x, 1, 0)$. We solve this problem in two steps:

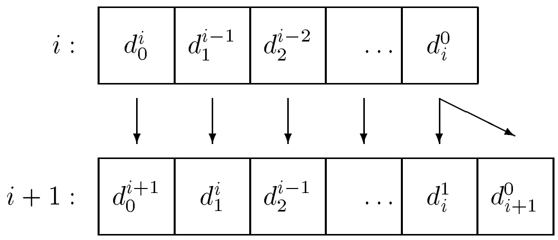


Figure 1.2 Computation of Derivatives of Total Order $i+1$

1. **(b, n) := derivatives(p)**

   Let $d_u^v \in \mathbb{Q}(x,z)$ denote $\frac{\partial p(x,1,z)^{u+v}}{\partial x^u z^v}$ of total order $u+v$. Our goal is to compute a sequence $[b_i]_{i=0}^n$ where b_0 is the greatest square free divisor (gsfd) of $p(x,1,0)$, $b_{i+1}(x)$ is the greatest common divisor (gcd) of $b_i(x)$ and of all $d_v^u(x,0)$ with $u+v=i+1$, and n is the smallest order such that $\deg b_n = 0$. The roots of $b_i(x)$ are the x-coordinates of all singularities $(x,1,0)$ of order at least $i+1$. The x-coordinates of all singularities $(x,1,0)$ of order i are thus the roots of $\frac{b_{i-1}(x)}{b_i(x)}$ for $1 < i \leq n$.

   The partial derivatives of total order $i+1$ are generated from (and overwrite) the derivatives of order i as illustrated in Figure 1.2:

   (b, n) := derivatives(p)
     $i := 0$
     $d_0(x,z) := p(x,1,z)$
     $b_0(x) := \text{gsfd}(d_0(x,0))$
     while $\deg(b_i) \neq 0$ **do**
       $d_{i+1}(x,z) := \frac{\partial d_i(x,z)}{\partial z}(x,z)$
       for j **from** 0 **to** i **do**
         $d_j(x,z) := \frac{\partial d_j(x,z)}{\partial x}(x,z)$
       $b_{i+1}(x) := b_i(x)$
       for j **from** 0 **to** $i+1$ **do**
         $b_{i+1}(x) := \text{gcd}(b_{i+1}(x), d_j(x,0))$
       $i := i+1$
     $n := i$
   end

2. **S := singularities(b, n)**

   Let q_i denote b_{i-1}/b_i for $2 \leq i \leq n$. Every root a of q_i is the x coordinate of a singularity $(a,1,0)$ of multiplicity i; S is the set of all such $(i,(a,1,0))$.

The first step accounts for most of the computation time of each curve transformation. Our goal is therefore to speed up **derivatives** by parallelization.

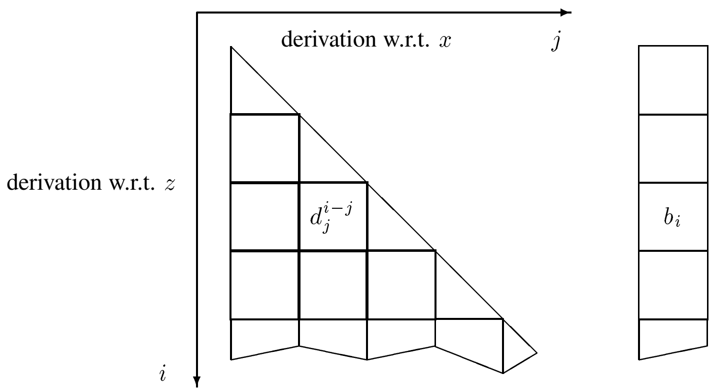


Figure 1.3 The Matrix of Partial Derivations

3. THE PARALLELIZATION APPROACH

Figure 1.2 suggests to organize the computation of all d_u^v in a triangular matrix shown in Figure 1.3: each line i contains all d_u^v with $u + v = i$, each column j contains all d_j^v, i.e., the matrix holds at position (i, j) the derivative d_j^{i-j}. The basic parallelization idea is to compute all those positions (i, j) with $i \geq j$ in parallel whose data dependencies have been resolved, i.e., for which $(i-1, j)$ is available (if $i > j$) respectively $(i-1, j-1)$ is available (if $i = j$).

However in order to increase its granularity, a task should compute multiple matrix elements, i.e., we have to *block* the computation accordingly. We achieve this by partitioning the triangular matrix into square blocks of size $m * m$ (for some blocking factor m) that comprise all partial derivatives that are computed by the same task. The blocks along the diagonal boundary of the triangular matrix are themselves triangular and only have to compute $\frac{m^2-m}{2}$ elements.

The tasks are created by a main program which itself computes iteratively the d_u^0, i.e., the derivatives along the diagonal boundary of the matrix. When it has computed all those derivatives that represent the diagonal boundary of a triangular block, it starts a corresponding task that computes the values of this block. Whenever a task (computing a triangular block or a square block) has terminated, the main program starts a new task for computing that square block that is adjacent to the lower boundary of the result block.

Actually, the result of a task need not be the values of all d_u^v in the corresponding block because we are only interested in

1. the last line of the block which is required by the task computing the adjacent block (this result need not be returned to the main program but can be put into a shared space from which the other task can retrieve it);

```
(b, n) := derivatives(p)
  T := ∅
  for i from 0 to m * p − 1 by p do
    compute d^0_{i+k}(x, z) for 0 ≤ k < m
    T := T ∪ {start(task, i, i)}
  end
  n := deg(p)
  while T ≠ ∅ do
    (t, T) := (any element of T, T − {t})
    (b', i, j) := wait(t)
    for k from 0 to m − 1 do
      if b_{i+k}(x) is initialized
        then b_{i+k}(x) := gcd(b_{i+k}(x), b'_k(x)) else b_{i+k}(x) := b'_k(x)
      if deg(b_{i+k}) = 0 ∧ i + k < n then n := i + k
    end
    enable (i + m, j)
    if i = j then enable (i + m, j + m)
    do
      select enabled (i, j) with minimum i and disable it
    while i ≥ n ∧ some block is enabled
    if i < n then
      if i = j then compute d^0_{i+k}(x, z) for 0 ≤ k < m
      T := T ∪ {start(task, i, j)}
    end
  end
  for i from 0 to n − 1 do b_{i+1}(x) := gcd(b_i(x), b_{i+1}(x))
end
```

Figure 1.4 The Parallel Algorithm

2. the greatest common divisor of each line of the block which is required to compute the greatest common divisor of the whole matrix line (this result is returned to the main program).

Since the gcd is commutative and associative, the program may receive in any order the results computed by the tasks of line i and combine them with the current value of b_i. In a final step, b_{i+1} is then combined with b_i.

To let the algorithm efficiently execute on a machine with a limited number of processors, we have to adopt an appropriate *scheduling strategy*. In particular, as soon as we have detected, for some i, the termination criterion $\deg(b_i) = 0$, only the results of those tasks are required any more that operate on lines with index less than or equal i. In order not to let superfluous tasks compete with required tasks for the p processors available, the main program does therefore not immediately start all tasks whose data dependencies have been resolved but schedules at most p tasks for execution at any time.

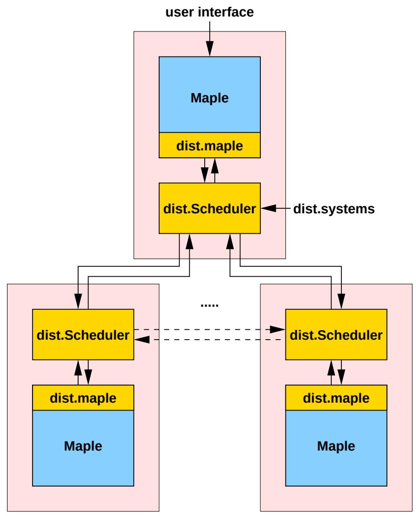


Figure 1.5 Distributed Maple Architecture

Initially, tasks are created for computation of the first p triangular blocks. Whenever a task terminates, we "enable" the adjacent square block whose computation thus becomes possible; if the terminated task has computed a triangular block (and it was not one of the p initial tasks), we also enable the subsequent triangular block. Among all enabled blocks, we choose a block with minimum line index (its computation may make the computation of blocks with larger indices superfluous). When the termination criterion is detected in line i, only those tasks will be started that operate on lines with indices less than i; when no more task is active, the algorithm terminates.

4. DISTRIBUTED MAPLE

Distributed Maple is an environment for writing parallel programs on the basis of the computer algebra system Maple [6]. It allows to create tasks and to execute them by Maple kernels running on various machines of a network. Each node connected to a session comprises two components (see Figure 1.5):

Scheduler The Java program `dist.Scheduler` coordinates node interaction. The initial scheduler process (created by the Maple kernel attached to the user frontend) reads all system information from file `dist.systems`; it then starts instances of the scheduler on other machines.

Maple Interface The file `dist.maple` read by every Maple kernel implements the interface between kernel and scheduler. Both components use pipes to exchange messages (which may embed any Maple objects in the compact linear format that Maple uses for library files).

After a session has been established, every scheduler instance accepts tasks from the attached computation kernel and schedules these tasks among all machines connected to the session. During the execution, additional socket connections between remote scheduler instances are created on demand.

The user interacts with Distributed Maple via a (text or graphical) Maple frontend. Maple commands establish a session in which tasks are created for execution on any connected machine, e.g.:

```
> read `dist.maple`;
Distributed Maple V1.0.7 (c) 1998 Wolfgang Schreiner (RISC-Linz)
See http://www.risc.uni-linz.ac.at/software/distmaple
> dist[initialize]([[gintonic,linux], [andromeda,octane]]):
connecting gintonic...
connecting andromeda...
> dist[all]("Integrate := proc(e, x) int(e, x) end:"):
> t1 := dist[start](Integrate, x^n, x):
> t2 := dist[start](Integrate, x^n, n):
> dist[wait](t1) + dist[wait](t2);
```

The file `dist.maple` implements the interface to the distributed backend as a Maple package `dist`. The command `dist[initialize]` asks the system to start the backend and create two additional Maple kernels on machines `gintonic` and `andromeda` of types `linux` and `octane`, respectively.

The following operations represent the core programming interface:

`dist[all](`*`command`*`)` lets the Maple statement *command* be executed on every Maple kernel connected to the session. If this command refers to a file, this file must be visible on every machine connected to the session.

`dist[start](`*`f, a, ...`*`)` creates a task evaluating *f(a, ...)* and returns a task reference *t*. Tasks may create other tasks; arbitrary objects (including task references) may be passed as arguments and returned as results.

`dist[wait](`*`t`*`)` blocks the execution of the current task until the task represented by *t* has terminated and returns its result. Multiple tasks may independently wait for and retrieve the result of the same task *t*.

This parallel programming model is essentially based on functional principles which is sufficient for many kinds of computer algebra algorithms [3]. In addition, the environment supports a non-deterministic form of task synchronization for speculative parallelism and shared data objects which allow tasks to communicate via a global store. The later facility was applied in the implementation of the algorithm presented in this paper for storing the last line of each block computed by one task and required as input for another task.

5. EXPERIMENTAL RESULTS

We have benchmarked the parallel variant of the derivatives computation with four examples for which the sequential computation takes 552s, 198s, 402s, and 1798s, respectively (the times refer to execution on a PIII@450MHz Linux PC). The examples were extracted from the resolution of the singularities of four artificially generated curves in which they represented the most time-consuming part (88%, 85%, 95%, and 95% of the total runtime). The program has been executed in two environments consisting of 24 processors each:

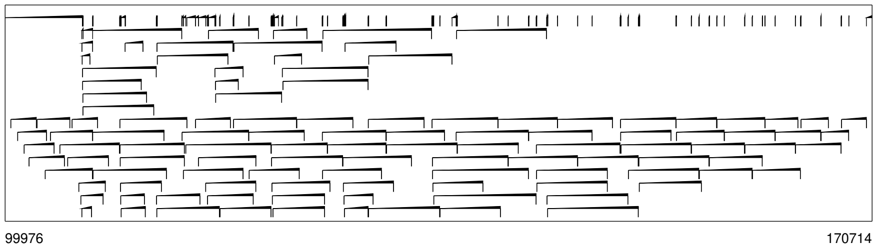
- a Silicon Graphics Origin multiprocessor (64 R12000@300 Mhz, 24 processors used) located at the university campus;
- a cluster of 4 SGI Octanes (2 R10000@250Mhz each) located in our institute and of 16 of the Origin processors; the Octanes are linked by two 10 Mbit Ethernet subnetworks connected via ATM to the campus.

The raw computing power of the Origin and of the cluster are 18.7 and 17.2, respectively; these numbers (measured by a Maple benchmark) denote the sum of the processor performances relative to a PIII@450MHz. In the cluster, the frontend was executed on an Octane such that the configuration behaved as if each Origin processor was executing on a separate workstation. We had to exclude our Linux PCs from the cluster, because their Java 1.1 did not support native threads, which severely downgraded the communication throughput of the multithreaded scheduler (we verified by a test installation that the forthcoming Java 2 port with native threads solves this problem). On the Origin, we achieved much better performance by using green threads.

The top diagram in Figure 1.6 generated from a Distributed Maple profile illustrates the execution of Example 3 in the cluster configuration with 8 Octane and 8 Origin processors (top and bottom half of the diagram, respectively). Each line represents a task being executed on one of the processors.

The table in Figure 1.6 lists the execution times in both environments for each input and varying numbers of processors. The subsequent row of diagrams visualizes the absolute speedups $\frac{T_s}{T_n}$ (where T_s denotes the sequential execution time and T_n denotes the parallel execution time with n processors), the second row visualizes the scaled speedups, i.e., the absolute speedups multiplied with $\frac{n}{\sum_{i=1}^{n} p_i}$ (where p_i denotes the performance of processor i), the third row visualizes the scaled efficiency, i.e., $\frac{T_s}{T_n \sum_{i=1}^{n} p_i}$. The scaled speedups and efficiencies of the Origin (marker $\times$) and of the cluster (marker $+$) are almost the same.

The scaled efficiencies with up to 4 processors are greater than one because of two effects: first, the new way in which the greatest common divisors of lines of derivatives are computed turns out to be more efficient for this input (which is a hint for a general algorithmic improvement); second, the overall increase of available Maple memory considerably reduces the garbage collection times.



Example	System	1	2	4	8	16	24
1 (552s = 88%)	Origin	560	294	190	113	83	80
	Cluster	817	436	233	155	100	101
2 (198s = 85%)	Origin	192	123	70	44	41	45
	Cluster	214	142	91	64	45	49
3 (402s = 95%)	Origin	371	218	128	70	59	62
	Cluster	518	265	153	107	71	73
4 (1798s = 95%)	Origin	2199	928	574	348	239	228
	Cluster	3015	1222	703	469	275	241

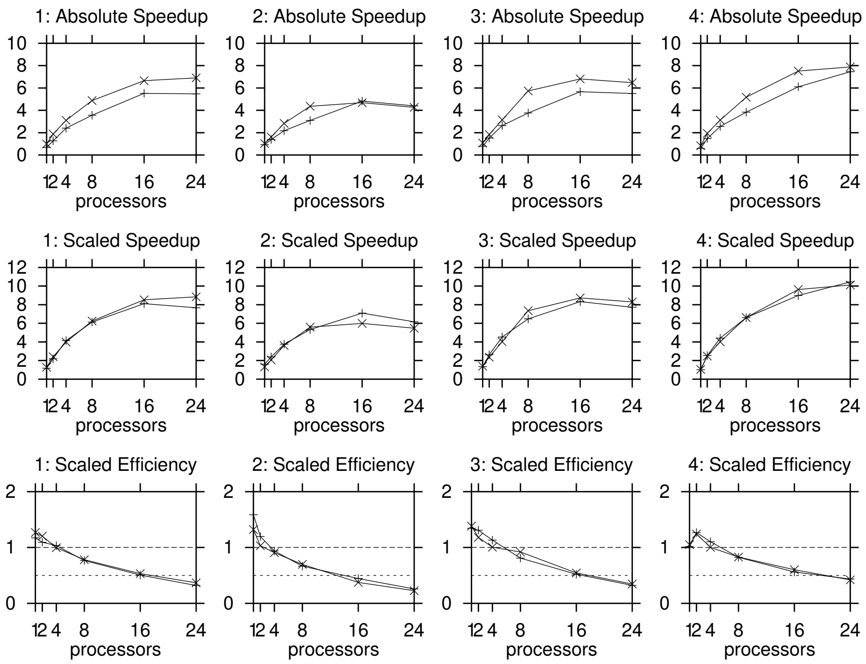


Figure 1.6 Experimental Results

6. CONCLUSIONS

We have demonstrated that also in distributed environments the parallel implementation of a computer algebra algorithm can achieve significant speedups which compete with those on a standalone massively parallel multiprocessor. We have faced super-linear speedups caused by subtle algorithmic changes and by reduced garbage collection times due to more memory available. The algorithm operates on large amounts of data, which made it necessary to carefully tune the parallelization: to select an appropriate blocking strategy and to place data into a virtually shared space. The implementation also exhibited some problems with the Java thread support on Linux PCs and on the Origin.

References

[1] L. Bernadin. Maple on a Massively Parallel, Distributed Memory Machine. In *PASCO'97 — Second Int. Symp. on Parallel Symbolic Computation*, pages 217–222, Maui, Hawaii, July 20–22, 1997. ACM Press.

[2] A. Diaz and E. Kaltofen. FoxBox: A System for Manipulating Symbolic Objects in Black Box Representation. In O. Gloor, editor, *ISSAC'98 Int. Symp. on Symbolic and Algebraic Comp.* ACM Press, New York, 1998.

[3] H. Hong, A. Neubacher, and W. Schreiner. The Design of the SACLIB/PACLIB Kernels. *Journal of Symbolic Computation*, 19:111–132, 1995.

[4] C. Mittermaier. Parallel Algorithms in Constructive Algebraic Geometry. Master's thesis, Johannes Kepler University, Linz, Austria, 2000.

[5] M. Mnuk and F. Winkler. CASA - A System for Computer Aided Constructive Algebraic Geometry. In *DISCO'96 — Int. Symposium on the Design and Implementation of Symbolic Computation Systems*, volume 1128 of *LNCS*, pages 297–307, Karsruhe, Germany, 1996. Springer, Berlin.

[6] W. Schreiner. Distributed Maple — User and Reference Manual. Technical Report 98-05, Research Institute for Symbolic Computation (RISC-Linz), Johannes Kepler University, Linz, Austria, May 1998.

[7] W. Schreiner. Developing a Distributed System for Algebraic Geometry. In B. H. Topping, editor, *EURO-CM-PAR'99 Third Euro-conference on Parallel and Distributed Computing for Computational Mechanics*, pages 137–146, Weimar, Germany, March 20-25, 1999. Civil-Comp Press.

[8] R. J. Sendra and F. Winkler. Symbolic Parametrization of Curves. *Journal of Symbolic Computation*, 12(6):607–631, 1990.

[9] K. Siegl. Parallelizing Algorithms for Symbolic Computation Using ||MAPLE||. In *Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 179–186, San Diego, California, May 19-22, 1993. ACM Press.